RNR-87-005

Remote Workstation Administration in a Supercomputing Environment

Eric Raible

June 11, 1987

Introduction

The Numerical Aerodynamic Simulation (NAS) facility [1] at NASA-AMES Research Center is composed of a heterogeneous collection of computers all running UNIX with TCP/IP. More specifically, the current machines are a Cray 2, two Amdahl 5840's, four Vax 780's, and thirty Silicon Graphics Iris workstations. A high-speed Network Systems Corporation (NSC) Hyperchannel network interconnects all of the machines. In addition, all machines except for the Cray have an ethernet interface.

This paper discusses our experiences in the system administration of the Silicon Graphics workstations in this environment. It begins with a chronological description of some of the problems which we encountered as network connectivity became a significant concern. Our early, naive approach to workstation administration is described, as are various problems we encountered. As a result of this experience, we were able to prototype some tools and techniques to solve the problems. Finally, the tools that are presently thought to be useful are described in some detail.

The Journey

The system administration of the workstations was generally considered to be a solved problem in January of 1986. At that point, all of our machines were running an unmodified Silicon Graphics (SGI) operating system, which included their XNS networking implementation. Although using their code "as-was" had the advantage of simplicity, it did not meet our needs at the time; a basic design goal of the NAS was that all machines run TCP/IP. It was because of this goal that we made so many changes to the Iris operating system. Unfortunately, these changes were completely incompatible to the work that SGI was doing at the time. In addition, TCP/IP requires many more programs and configuration files that XNS. For these reasons we were forced to take system administration more seriously.

The ad hoc approach

Perhaps the most memorable aspect of the NAS in early 1986 was the unreliability of the hyperchannel and ethernet networks. Most of these problems stemmed from the fairly radical idea of having TCP/IP on both hyperchannel and ethernet on every machine (except the CRAY2, which has only hyperchannel). As a direct consequence of this decision, every machine on the network was undergoing extensive modifications in order integrate these two quite different network interfaces into its operating system. In addition to the development on each machine, there was also the non-trivial problem of agreement on and coordination of the multitude of details of data representation inherent in multi-machine communication.

As a result of these changes all happening at once, network communications were not very stable. Things that worked one day were hopelessly broken the next. Software that was known-good became non-functional due to seemingly unrelated changes on other machines. Inconsistencies in configuration files caused much confusion, and the occasional hardware failure became a major chore to track down. But there is an ironic twist to all of this — the one thing that made it possible to use the network at all was precisely the source of all the problems: since every machine had two completely separate network interfaces, it was generally possible to find a path through the network that worked well enough. Finding that path, however, was time consuming and frustrating. Transferring a single file typically took half a dozen tries using different routes.

Given this situation, it is not surprising that almost all of the system support on the NAS was done on hard-wired terminals. This worked well enough for all of the machines except the workstations. First, there was the issue of geographic inconvenience: the Iris terminals were spread over three floors in two separate buildings. In addition, these terminals were already heavily used. As a result the network was the only feasible way to maintain the workstations in a consistent and efficient manner. Given the network problems described above, this was harder than it sounds. The straightforward shell-scripts [2] which were tried at first could not deal with errors gracefully. Next, a slightly more complex strategy that included retry on failure was attempted. This had mixed results; often a new version would be ready before the last version was fully distributed. There were countless situations in which an intermittent networking failure between workstations was eventually tracked down to incompatible versions of some piece of software.

As if this weren't bad enough, we were also tracking the official SGI releases because some of the user's required XNS. Typically, there would be one or two XNS and three or four TCP/IP operating systems running on different machines. To a large degree, the software that was running on any particular machine was dictated by the desires of the primary user's of that machine; once they got used to the quirks of a particular version, there was reluctance to upgrade to something "better". Of course, this complicated administration, and made some networking bugs impossible to track down.

The only way to get a handle on this situation was to log onto each machine in turn, and verify that everything was in order. The network itself, and the network software on the workstations, was too unreliable to allow any other way. The only reason that it was tolerable was that we fully expected it to eventually get better.

The Transition to Order

In retrospect, it is clear that many of the difficulties described above were the result of our enthusiasm to get TCP/IP running on the workstations as soon as possible. The XNS-based communication software which worked quite reliably on the IRIS was incompatible with the rest of the NAS. But as that time, it was crucial to integrate the workstations into the rest of the network, both on hyperchannel and ethernet.

Of course, this eventually happened. As the software improved, it became less critical to install the latest fix as soon as it was done; the previous version would suffice until the new one was fully tested. This was especially true of new kernels, which simplified life considerably. And because there were fewer "fires", we had the time to develop a set of tools to help manage the various version of the operating system. At one time, we could install one of four or five completely different versions of the operating systems with one command. This was invaluable for testing, development, and migration of new software to user machines.

As the standard TCP/IP utilities (rsh, rcp, rlogin, telnet, ftp) became more robust, it was possible to create shell scripts to keep various machines up-to-date. Since there were so many fewer possible failure modes, it became possible to deal with all of them. At one point, we decided to distribute an entirely new operating system over the network by creating a huge archive, and expanding it automatically on every machine using a shell script. Of course, after the expansion, there were many customizations that had to be performed on each machine; these were handled automatically by the shell script. This approach worked quite well, and had many benefits related to the fact that it enforced the restriction that all machines ended up looking exactly the same.

Although this resynchronization worked well, the shell script which controlled it was quite complex. For example, when replacing *every* system file on the machine, it was necessary to take great pains to avoid errors which would have left the workstation in an inconsistent state. Otherwise, there are scenarios in which the machine would have to be restored from tape... but our Iris's did not have tape drives! There was also the problem of replacing those programs which were running while the shell script was executing; these had to be moved out of the way and deleted once the expansion was successfully completed. So although this endeavor was considered a great success, the shell script in question was far too large and fragile. It was clear that a more general solution would eventually be needed.

At this point, all of the machines looked exactly alike, and new versions of programs were being released at a sane rate. For obvious reasons, it became important to manage changes in a well-defined fashion to avoid the earlier chaos. So we divided the workstations as follows: three for development, twenty for users (with two of these also used for testing), and one template machine. The test machines belonged to the more sophisticated, adventurous users. The template machine could be used to rebuild any of the user machines. Software migrated from development to the users via the test machines. There were two golden rules: 1) don't install anything before the the test-users give their okay, and 2) always install software on the template before distributing it to the users. Because we backed up machine-specific files to the template, we could then rebuild any user machine in case of disaster.

Of course there were shell scripts to perform whatever commands were necessary on each of the twenty user machines, but these were too slow. A more sophisticated error-logging batch-oriented system was developed, but this solved the wrong problem. An *interactive* approach to maintaining our consistency was needed. To achieve the necessary speed, all of the commands had to execute in parallel. A tool to manage parallel execution of commands on remote machines was developed, and is discussed below.

In addition to this sort of parallel execution, a specialized tool was needed to perform remote file distribution. In order to claim that some arbitrary set of workstations were running the same operating system, literally hundreds of files would have to be the same. This is clearly too much to manage by hand. And although TCP/IP provided the necessary foundation for such a tool, that tool had yet to be developed. The important design criteria were simplicity of use, sufficiently powerful, and programmable to allow it to handle unforeseen problems. Luckily for us, the folks at University of California at Berkeley were developing such a tool, which we were able to tailor for our own use.

The Tools

Concurrent Command Invocation

The Berkeley version of Unix includes the "rsh" program, which executes a single command on a remote system. What was needed, however, was the ability to execute a *series* of commands on a *group* of remote systems. This, combined with the an interactive interface, we felt had the makings of a very powerful tool. The tree-shell, a program derived from the Berkeley rsh. [1] The tree-shell ("trsh") is so named because it distributes the communication channels across multiple machines in a tree-like structure. For instance, the top level trsh might be directly communicating to six intermediate machines. Each of these, in turn, might be connected to five others. Organizing things in this was has two benefits: it avoids the Unix limitation on the number of connections in a single process, and it reduces the network impact of having thirty machines simultaneously sending their responses to the top level machine. In addition, there is a small amount of processing at each intermediate node, which helps distribute the computational burden.

In a scheme such as this, presenting the output from a large number of machines in an

[1]Tree-shell was written by David Tristram, a developer on the NAS project. Although he started with rsh, the changes have been quite extensive (available from the author).

effective manner is a major concern. Trsh handles this problem in three ways: the machine name precedes the output from every machine, this output is always presented in the same order (specified at invocation time), and the top level trsh pages the output so that results don't scroll of the screen unexpectedly.

The synchronization mechanism used to present the output in machine order is unexpectedly elegant. Since it is possible to determine when the connection to a remote machine is closed (deliberately or due to some failure), it is feasible to assume that every machine with an open connection will eventually respond. In other words, it will eventually produce some output, and then print print out a prompt. So for each machine with an open connection, trsh prints out the machine name, and copies the output from that machine to the screen until it reads a prompt. It then goes on to the next machine. In most cases this happens extremely quickly, since all the machines are done and waiting by the time the first machine is handled.

The tree-like connection structure of trsh can nest to any depth as follows. Since trsh is a program that can be executed like any other, trsh simply invokes another trsh on each of its children (recursively) to create a tree of connections. Each invocation of trsh collects the output from its children, and prints it out for its parent. The command-line specification of this connection tree is recursively defined, and in practice, is trivial to use.

The one remaining essential feature is selective invocation of commands depending on the machine name. By default, every command gets executed on every machine. The method for restricting this is quite simple. Upon startup, several aliases are defined for every remote shell (in the C-shell aliases are simply command level macros). Among these are *on* and *not-on*. In operation, *on* compares its first argument with the current machine name. If it matches, the rest of the arguments are taken as a command and executed. Otherwise it does nothing, and a prompt is printed out. Since these macros (like all commands) are executed on all machines, the effect is as desired. *Not-on* is analogous, and won't be described. This design is a good example of taking advantage of existing capabilities of a widely used program (csh).

There are two other important macros: "hold" and "release". These control whether the output from a command will be printed or not. They work as follows: "hold" sets the prompt to nil, and "release" sets it back to the default. Since the prompt is used to decide when to print output (as described above), these also work as advertised. The main use of these macros is to perform a multi-command task, with the output from just the first machine being displayed, until it is desired to see the results from all of the other machines.

Remote File Distribution

Rdist [3], a program for remote file distribution, is part of 4.3 Berkeley Unix Its purpose is to maintain identical copies of files over multiple hosts. Rdist provides a standard "client/server" model, where the "client" machine distributes files to the "server" machine(s). With its local enhancements [2], which are described only as part of the overall description below, rdist has proved to be a most useful tool for many tasks, and especially for workstation administration. It has sensible defaults, an orthogonal set of options, and a "language" well-suited to its stated task.

A concept basic to the operation of rdist is to only distribute files which *need* to be distributed. This is accomplished by comparing the local and remote values of five particular file attributes: the last modification date, the size of the file in bytes, the ownership of the file, the permission bits on the file, and optionally, a full binary comparison. Rdist gathers this information on the client machine by simply reading it off the disk, and on the server machine by requesting the information over a network connection. Only if any of these attributes differ is the file transferred to the remote machine.

Once a file is transferred, these attributes will be identical. This has important ramifications when updating a large number of files: if for some reason rdist is interrupted, it can continue where it left off with only minimal overhead (comparing the few bytes of attribute information per file). Using these attributes also allows rdist to run in "verify" mode, where instead of actually updating files, it simply prints out what it *would* update. In addition to providing a degree of safety, this option is invaluable in tracking changes to various machines.

There are several features of rdist which make it particularly appropriate for the system administrator who must be concerned about a large number of machines. Most importantly, it enables him to automatically keep machines up-to-date, and to track changes. For instance if unauthorized changes are made in system directories, rdist will report this; it can be investigated. There is no other practical way to check such things on 30 machines. In addition, there is the capability to remove any file on the remote machine which does not exist on the local machine. This ensures that both machines will be *exactly* identical. In essence, allows the system administrator to configure just the template machine, and then clone any subtree of that of the template to any other machine trivially. Since rdist even takes care of updating running programs, there is no special preparation necessary.

Rdist transparently handles many of the details associated with the common case of installing few files on one remote machine. If any of the files are directories (in Unix, directories are simply a special type of file), rdist simply distributes the contents of the directory recursively. If the necessary directories do not exist on the target, they are created automatically. If the files are absolute pathnames, they are installed in the same place on the remote machine; if not, they are installed relative to the users home directory. Sensible defaults such as these make *any* program easier to use, and rdist is no exception.

Rdist was not designed in a vacuum - where functional similarities exist between rdist and other Unix programs, an attempt was made to make rdist to conform to the existing de facto standards. As a result, rdist uses the same network permission structure as the

[2]Available from the author; rdist has been ported to the Silicon Graphics Iris release 3.5, an Amdahl 5840 running UTS, and the Cray 2 running Unicos 2.0.

other Berkeley commands (rcp, rsh, rlogin). Since the NAS project has adopted these conventions, rdist was easily integrated into our environment. The choice of command-line options was obviously influenced by other programs. And the similarities between make [4], which is most often used to compile only those source files which need recompilation, and rdist, which is used to distribute only those files which are out-of-date, are striking. All in all, we have found that rdist is as indispensable in the area of system administration as make is for programming large projects.

In order to harness the full power of rdist, it is necessary to write a "distfile". This is easy to do - the syntax of a distfile is fairly intuitive. There are three major capabilities (in addition to all that has been discussed so far) which distfiles provide: they allow the specification of files to ignore in its operation; they list arbitrary shell commands to be performed after installation; and they support distribution to multiple machines.

There are two types of commands basic to the specification of the files of interest: *install* commands, and *except* commands. The former can use the standard C-shell wildcard characters to specify an arbitrary list of files: all of the FORTRAN files in a particular directory, for instance. The later use the standard Unix regular expression syntax (ala ed or vi) to specify which files to ignore. Taken together, these commands allow for a fairly general approach towards solving the problem of file specification.

In many cases, it is desirable to combine file installation with a few commands after the installation. The fact the the name of the file being updated is available to commands running on the remote machine makes this a very general mechanism. As an example, a makefile for a particular program which runs on many different machines might rdist the source to one (test) machine of each type. The relevant distfile could then run make on each of those machines, thereby ensuring that the latest version is always ready to test on every machine. A technique very similar to this is in fact used to maintain rdist itself on four different types of machines in the NAS.

The default for rdist is to update all of the machines listed in the distfile. It is easy, however, to limit the changes to affect only a subset of these machines (for debugging, etc) using command line options. Once the distfile is completely debugged, the normal mode is to update all of the machines, which takes one command: "rdist".

There are several minor features of distfiles which are worth mentioning. A macro facility allows for shorter, more readable distfiles. It is possible to redefine any definition in the distfile from the command line, which can be helpful in some circumstances. In addition, it is possible to have labels, which can be jumped to, again, by command line specification. And finally, it is possible to list a group of people to receive an electronic mail message concerning the result of using that distfile.

Below is an example distfile, which is intended to be self-documenting.

```
#
# Example distfile
```

```
#
HOSTS    = ( igor rodan ernst chewy wk05 )
BINARIES = ( gsh bindkey )
MANUAL   = ( gsh.1w )

$(BINARIES) -> $(HOSTS)
    # Built in a source directory,
    # installed in /usr/local/bin
    install /usr/local/bin;
    # rdist will preserve ownership and
    # permissions, but these are
    # owned by me...so we set them here.
    special "chown root /usr/local/bin/gsh;
             chmod 4755 /usr/local/bin/gsh";

$(MANUAL) -> $(HOSTS)
    install /usr/man/u_man/man1/gsh.1w;
    # Remove the formatted one so that the
    # new one gets formatted
    special "rm /usr/man/u_man/cat1/gsh.1w.z";

/usr/local/bin -> $(MACHINES)
    # Make /usr/local/bin identical
    # -R means remove any remote files
    # which don't exist here
    install -R;
    # These are only licensed for me
    except ( di3000 franz-lisp );
    # Send me mail.
    notify raible@ames-nas.arpa;
```

Payoffs of These Tools

It is possible to think of the services offered by Unix as manipulating just two types of objects: files and processes. It is perhaps because of this simple model that rdist and trsh complement each other so well: rdist excels at managing large numbers of files, while trsh was designed to control multiple process simultaneously. With respect to system administration, rdist is used to configure remote machines, while trsh can be used to control them. In any case, they work well together, and solve most of our problems.

As a result, instead of chaos, we are moving towards a more integrated environment.

Not only do all (user) workstations run the same software, but distributed applications are becoming more and more common. This would not have been possible in the early chaotic environment, but is an important point: in a very real sense, interactive and distributed computing is what the NAS is all about.

The tools discussed here are also being used in areas other than workstation administration. Rdist is used to keep the four Vaxes in sync, mostly because the concept has been proven on the workstations. It is also used to maintain machine-independent data-bases, such as Macsyma and emacs libraries, as well as TEX fonts on the different NAS machines. Trsh has been used to purposefully stress the hyperchannel network by simultaneously copying a large file from ten workstations to the Cray. This is useful for characterizing traffic-induced bugs, and performance evaluation. Since rdist works so well with make it is being used by developers to maintain programs on different machines because it eliminates extraneous copying (due to rdist) and compiling (due to make). And finally, we are experimenting with using rdist for backup to our mass storage system [3].

Of course, there are many other tools useful for these tasks, as well as for workstation administration, but rdist and trsh are two that we have found to be particularly helpful. Perhaps part of our success with these two programs is due to the fact that rather than trying to do everything from scratch, they build upon the Unix environment. It is because of this symbiosis that they not only perform admirably well in the areas for which they have been designed, but have found other uses as well.

References

[1] BAILEY, F. R. Status and Projection of the NAS Program. NASA Technical Memorandum 88339, 1986.

[2] BOURNE, S. R. The Unix Shell. *The Bell System Technical Journal 57*, 6 (1978), 1971–1990.

[3] COMPUTER SCIENCE DIVISION, D. O. E. E., AND SCIENCE, C. *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution.* University of California at Berkeley, Berkeley, California, 1983.

[4] FELDMAN, S. I. Make — a Program for Maintaining Computer Programs. *Software Practice and Experience 9*, 4 (1979), 255–265.

[3]Thanks to my colleague Creon Levit for his help in this area